CSCG 2025 - KDF_dream

Challenge Information

Description:

> We've managed to insert ourselves into a secure channel between two covert agents, however we overplayed our hand and they have become suspicious that their channel is compromised.
>
> Realising that there is no way to restablish trust over the compromised network, Alice called for them to carry out a NIST Certified KDF protocol to generate a symmetric OTP, and then for them to use this to encrypt a physical message at a dead drop location.
>
> We want to control the message she leaves, can you influence their conversation to control what Bob reads at the dead drop?

- Category: `Crypto`
- Difficulty: `Medium`
- Author: `KILLERDOG`

Challenge Setup

As mentioned in the description, we as the attacker are the man-in-the-mittle and can read and modify Alice and Bobs messages.

Alice and Bob follow the following protocol:

- Diffie-Hellman (DH) key exchange
- SHA256 digest of the shared secret (in base 10) is `shared_key`
- Agree on Key-Derivation-Function (KDF)
- Pick 16B random nonce each
- Exchange nonces
- concatenate nonce parts
- Derive OTPs using SP800_108_Counter, the shared key, the nonces and the KDF

After the execution of this key agreement protocol, Alice uses its OTP to encrypt `"wearecompromised"` and stores it somewhere for Bob to access, but unreachable to the MiM. Bob decrypts the message and checks whether it is `"allgoodprintflag"` and prints the flag, else fails with an error.

Crypto Primitives

KDFs

The challenge offers the usage of the following three KDFs, to transform a message $m = m_1, \ldots, m_n$ with key k to a pseudo-random 16 byte string.

- `CMAC_PRF`:

```
def CMAC_PRF(K, M):
if len(K) == 16:
    K = K
else:
    K = CMAC.new(bytes(bytearray(16)), K, AES).digest()
PRV = CMAC.new(K, M, AES).digest()
return PRV
```

  The next part of the explanation is based on the wikipedia page for CMAC.
  The setup of CMAC is:
  - Encrypting the zero block with AES and k: $k_0 = E_k(0)$
  - if $msb(k_0) = 0$:
    - $k_1 = k_0 \ll 1$
  - else:
    - $k_1 = (k_0 \ll 1) \oplus C$, with $C = 0x87$ for 16B blocks
  - if $msb(k_0) = 0$:
    - $k_2 = k_1 \ll 1$
  - else:
    - $k_2 = (k_1 \ll 1) \oplus C$, with $C = 0x87$ for 16B blocks
  - If there is a incomplete block m_n:
    - $m'_n = k_2 \oplus (m_n \parallel 10\ldots0_2)$
  - else:

- $m'_n = k_1 \oplus m_n$

Then the tag is calculated by encrypting $m_1, \ldots, m_{n-1}, m'_n$ sing AES-CBC and k.

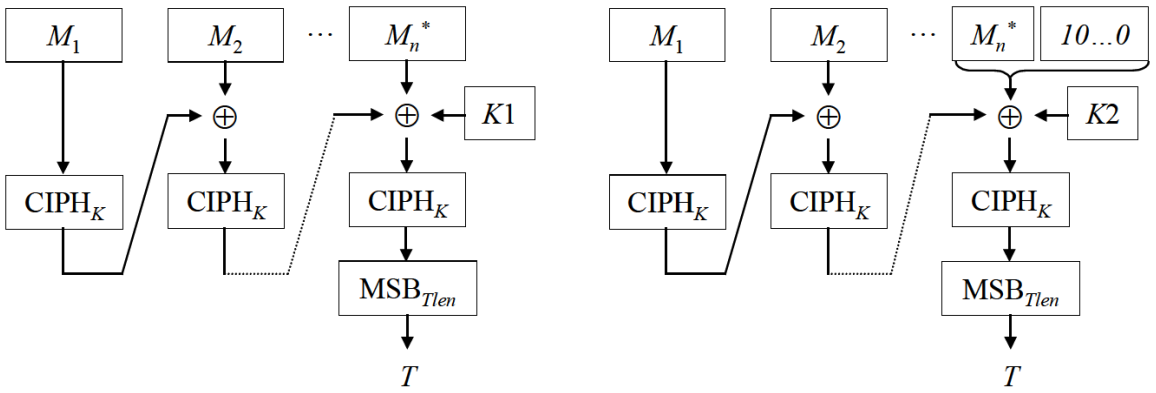


Figure 1: Illustration of the two cases of MAC Generation.

(img-source: https://i.sstatic.net/ISalk.png)

- `HMAC_PRF` :

```
def HMAC_PRF(K,M):
PRV = HMAC.new(K,M,SHA256).digest()[:16]
return PRV
```

We can see in the code: HMAC uses SHA256. As it is of no further interest to the solution, we will not go deeper into HMAC.

- `KMAC_PRF` :

```
def KMAC_PRF(K,M):
    PRV = KMAC128.new(key=K, data=M, mac_len=16).digest()
    return PRV
```

KMAC is based on KECCAK, the core SHA-3 algorithm. As it is of no further interest to the solution, we will not go deeper into the details.

SP800_108_Counter

The SP800_108_Counter algorithm is used to derive one or more keys from a master secret. It requires multiple parameters:

- the `master` key
- the length of the derived keys, `key_len`
- the pseudo-random function (`prf`) (here: one of the KDFs)
- the number of keys to generate, `num_keys`
- a byte string `label`
- a byte string `context`

For this challenge, we have:

- `master` = `k_{alice}` or `k_{bob}`
- `key_len` = `16`
- `prf` ∈ `{CMAC, HMAC, KMAC}`
- `num_keys` = `1`
- `label` = `b"keygen_for_secure_bagdrop"`
- `context` = `nonce`

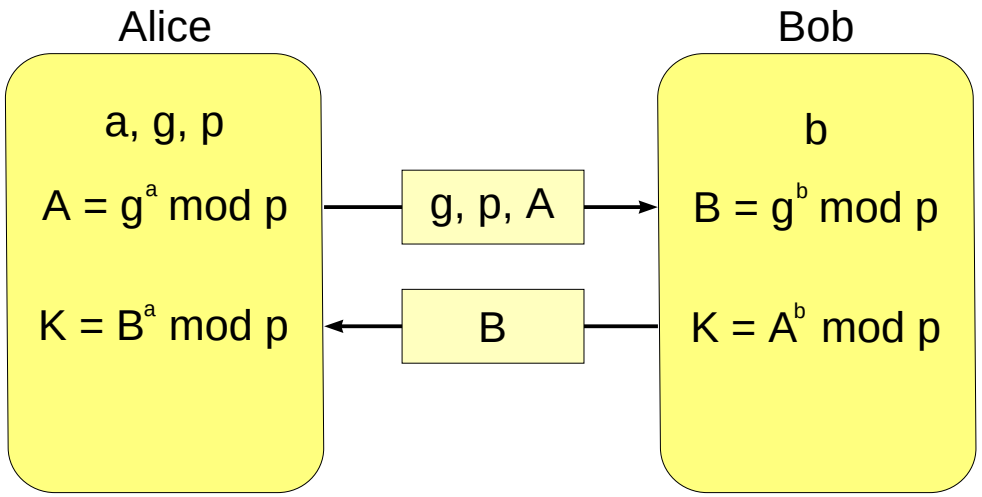
The standard defines that `prf` is repeatedly called with the `master` key and a `info` byte string until enough key material has been generated. The `info` is constructed by the concatenation of the following parts:

- a counter `i` , as 4 byte big-endian number (initialized to `1` and incremented for every call to `prf`)
- `label`
- `b"\x00"` , a zero byte
- `context`
- the total length of generated key material in bits as 4byte big-endian number ($key\_len \cdot num\_keys \cdot 8$)

Exploit

The target of the exploit is to modify the exchanged data in a way that `alice_OTP ^ 'wearecompromised' ^ 'allgoodprintflag' = bob_OTP` . That means we have

to find a preimage for the computed OTP.



$K = A^b \bmod p = (g^a \bmod p)^b \bmod p = g^{ab} \bmod p = (g^b \bmod p)^a \bmod p = B^a \bmod p$

The first thing to do, is to use MiM to exchange keys with Alice and Bob, so we get to know the shared key. If you don't know how DH works, look at the wikipedia article for it. Also, above the is the schematic for Diffie-Hellman from wikipedia.
For simplicity, we use the same private secret for both parties. To be precise, we select our own secret c, calculate $C = G^c \mod P$ and send that to both parties. With the intercepted A and B, we can calculate $K_{alice} = A^c \mod P$, $K_{bob} = B^c \mod P$, which are the shared secret between the MiM and either Alice or Bob. In general: $K_{alice} \neq K_{bob}$. To get the keys, k_{alice} and k_{bob}, we can do the following calculation. Below its shown for Alice, its analogous for Bob.

```
k_alice = SHA256.new(str(k_alice).encode()).digest()[:16]
```

In the following we have to be careful, which key to use for what.

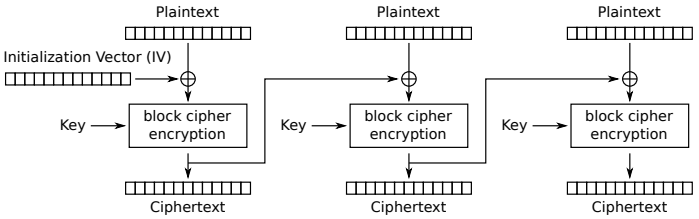
- k_{alice}: used to calculate OTP_{alice}
- k_{bob}: used to calculate OTP_{bob} and find `bob_ctx_1`

The challenge offers three KDF algorithms and we need to choose which one to use. As `KMAC` and `HMA` are based on secure hash functions / constructions, it is not computationally feasible to find a preimage. Therefore, we want to use `CMAC` because `AES-CBC` is easily reversible.

The only thing we can influence in the OTP calculation is the nonce, apart from the KDF, of course.
As both parties select one part of the nonce, exchange it and then concatenate the two parts, we can only influence one half in the calculation. For example, Bobs nonce: $bob\_ctx = bob\_ctx_1 + bob\_ctx_2$. To be precise, we want to modify bob_ctx_1 which is sent from Alice to Bob. We know bob_ctx_2 because it was sent to Alice previously but we cannot modify it on Bob's side.

We know from the chosen parameters for SP800_108_Counter that CMAC will only be called once, because a 16B key is requested and CMAC produces 16B of key material. The length of the `info` is 66B and the nonce part 1 (`bob_ctx_1`) is at `info[30:46]` . Unfortunately, this does not align with block boundaries, but it nonetheless allows us to find `bob_ctx_1` .

Since the `bob_ctx_1` is in the middle of `info` , we have to do two phases. In the first phase, we do AES-CBC forwards until we know the ciphertext of m_2: c_2.
In the second phase, we go backwards until we now $m_3 \oplus c_2$. By simply xoring c_2 and $m_3 \oplus c_2$, we get `bob_ctx_1[2:16]` :



Cipher Block Chaining (CBC) mode encryption

To find the remaining two bytes, I simply used a brute force over all possible values for `bob_ctx_1[:2]` and then checked that the last two bytes of $m_3 \oplus C_2$ equal `bob_ctx_2[:2]` . Because all the bytes in the ciphertext are dependent on all plaintext bytes in AES, we cannot simply calculate them.

Mitigation

The most direct fix rendering the attack impossible is to remove CMAC from the allowed KDFs. Hashing based KDFs should usually be preferred.

Hashing the nonce before using it, as was done with the shared secret to generate the shared key, would also render the attack impossible, because one has to find the preimage, which is computationally infeasible for a secure hash-function.

To stop the attack from manipulating the exchanged nonces, a Message Authentication Code (for example by using the shared secret with HMAC) could and should be used. To stop the attack from the reading said values, encryption with the shared key is the way to go.

Using certificates for the DH key exchange (to provide authentication) would also make the attack impossible because the attacker can no longer manipulate the exchanged values. The Station-to-Station (STS) protocol solves this problem.

Flag

`dach2025{But_n1st_said_it_was_fine?!???_15f7a069}`

Solve Script

```python
from primitives import CMAC_PRF, SHA256
from Crypto.Protocol.KDF import SP800_108_Counter
from Crypto.Cipher import AES
from Crypto.Util.number import long_to_bytes, bytes_to_long
from pwn import *

def xor(a, b):
    assert len(a) == len(b)
    return bytes([x^y for x,y in zip(a,b)])

def find_ctx(target_OTP: bytes, second_ctx_part: bytes, bob_key: bytes) -> bytes:
    for first_byte in range(1, 256):
        for second_byte in range(1, 256):
            first_bob_ctx1_part = bytes([first_byte, second_byte])
            ecb = AES.new(bob_key, AES.MODE_ECB)
            const_Rb = 0x87 # 16B block size
            zero_block = b"\x00" * 16
            L = ecb.encrypt(zero_block)

            def _shift_bytes(bs, xor_lsb=0):
                num = (bytes_to_long(bs) << 1) ^ xor_lsb
                return long_to_bytes(num, len(bs))[-len(bs):]

            if L[0] & 0x80:
                k1 = _shift_bytes(L, const_Rb)
            else:
                k1 = _shift_bytes(L)

            if k1[0] & 0x80:
                k2 = _shift_bytes(k1, const_Rb)
            else:
                k2 = _shift_bytes(k1)

            cbc = AES.new(bob_key, AES.MODE_CBC, zero_block)

            label = b'keygen_for_secure_bagdrop'
            assert len(label) == 25

            # known blocks before bob_ctx1[2:16]
            known_start_blocks = [
                long_to_bytes(1, 4) + label[:12],
                label[12:] + b"\x00" + first_bob_ctx1_part
            ]

            padding = b"\x80" + b"\x00" * 13 # info is 66 bytes long, pad to 80 bytes, i.e. next multiple of 16B
            key_len_enc = long_to_bytes(16 * 1 * 8, 4)

            # known blocks after bob_ctx1[2:16], in reverse order
            known_end_blocks = [
                key_len_enc[2:] + padding,
                second_ctx_part[2:] + key_len_enc[:2]
            ]

            # run cbc forward until context bytes
            last_ct = zero_block
            for data_block in known_start_blocks:
                last_ct = cbc.encrypt(data_block)

            # run backwards from expected mac / OTP
            backward_pt = ecb.decrypt(target_OTP)
            assert(len(known_end_blocks) == 2)
            # unmangle last (incomplete) block and get the IV / ciphertext of the previous block
            backward_last_ct = xor(xor(backward_pt, k2), known_end_blocks[0])
            backward_last_ct = AES.new(bob_key, AES.MODE_ECB).decrypt(backward_last_ct)
            backward_last_ct = xor(backward_last_ct, known_end_blocks[1])

            backward_input = AES.new(bob_key, AES.MODE_CBC, last_ct).decrypt(backward_last_ct)

            if backward_input[14:16] == second_ctx_part[:2]:
                return first_bob_ctx1_part + backward_input[:14]
    return None

# local
# p = process(["python", "chal.py"])
```

```python
# remote
p = remote("<session-id>-9999-kdf-dream.challenge.cscg.live", 1337, ssl=True)

# 2048 bit DH group: https://www.ietf.org/rfc/rfc3526.txt
P = 0xFFFFFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD129024E088A67CC74020BBEA63B139B22514A08798E3404DDEF9519B3CD3A431B302B0A6DF25F14374FE1356
G = 2

# free to choose
fake_alice_x = 2
fake_bob_x = fake_alice_x

p.recvuntil(b"msg: ")
alice_a = int(p.recvuntil(b"\n", drop=True))
alice_b = pow(G, fake_alice_x, P)
p.sendlineafter(b"to receive?", str(alice_b).encode())
p.recvuntil(b"msg: ")
bob_a = int(p.recvuntil(b"\n", drop=True))
bob_b = pow(G, fake_bob_x, P)
p.sendlineafter(b"to receive?", str(bob_b).encode())

bob_key = SHA256.new(str(pow(alice_a, fake_alice_x, P)).encode()).digest()[:16]
alice_key = SHA256.new(str(pow(bob_a, fake_bob_x, P)).encode()).digest()[:16]

kdf_protocol = "CMAC"
p.sendlineafter(b"to receive?", kdf_protocol.encode())
p.sendlineafter(b"to receive?", kdf_protocol.encode())

p.recvuntil(b"msg: ")
bob_ctx2 = bytes.fromhex(p.recvuntil(b"\n", drop=True).decode())
alice_ctx2 = bob_ctx2
p.sendlineafter(b"to receive?", alice_ctx2.hex().encode())

p.recvuntil(b"msg: ")
alice_ctx1 = bytes.fromhex(p.recvuntil(b"\n", drop=True).decode())

alice_ctx = alice_ctx1 + alice_ctx2

orig_msg = b'wearecompromised'
target_msg = b'allgoodprintflag'

alice_OTP = SP800_108_Counter(alice_key, 16, CMAC_PRF, 1, b'keygen_for_secure_bagdrop', alice_ctx)

enc_msg = xor(orig_msg, alice_OTP)
target_OTP = xor(enc_msg, target_msg)
bob_ctx1 = find_ctx(target_OTP, bob_ctx2, bob_key)
print(bob_ctx1)
assert bob_ctx1 is not None

bob_ctx = bob_ctx1 + bob_ctx2

# just to check
assert len(bob_ctx) == 32
bob_OTP = SP800_108_Counter(bob_key, 16, CMAC_PRF, 1, b'keygen_for_secure_bagdrop', bob_ctx)

print(target_OTP, bob_OTP)
print(xor(enc_msg, bob_OTP))

p.sendlineafter(b"to receive?", bob_ctx1.hex().encode())

p.interactive()
```